
UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LEONARDO TIRONI FASSINI

# SISTEMA OPERACIONAL EM TEMPO REAL FREERTOS

ANÁLISE DOS COMPONENTES DE *TIMERS*, INTERRUPÇÕES E GERENCIAMENTO DE RECURSOS NA PLATAFORMA DE HARDWARE MINIMALISTA ARDUINO UNO

CHAPECÓ
2018

**LEONARDO TIRONI FASSINI**

# SISTEMA OPERACIONAL EM TEMPO REAL FREERTOS

ANÁLISE DOS COMPONENTES DE *TIMERS*, INTERRUPÇÕES E GERENCIAMENTO DE RECURSOS NA PLATAFORMA DE HARDWARE MINIMALISTA ARDUINO UNO

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Prof. Dr. Marco Aurélio Spohn

**CHAPECÓ**
**2018**

Fassini, Leonardo Tironi

Sistema Operacional em Tempo Real FreeRTOS / Leonardo Tironi Fassini. – 2018.
45 f.: il.

Orientador: Prof. Dr. Marco Aurélio Spohn.
Trabalho de conclusão de curso (graduação) – Universidade Federal da Fronteira Sul, curso de Ciência da Computação, Chapecó, SC, 2018.

1. FreeRTOS. 2. Arduino Uno. 3. Timer. 4. Mutex. 5. Interrupção. I. Spohn, Prof. Dr. Marco Aurélio, orientador. II. Universidade Federal da Fronteira Sul. III. Título.


E-mail: leehtironi@gmail.com

**LEONARDO TIRONI FASSINI**

**SISTEMA OPERACIONAL EM TEMPO REAL FREERTOS**
ANÁLISE DOS COMPONENTES DE *TIMERS*, INTERRUPÇÕES E GERENCIAMENTO DE RECURSOS NA PLATAFORMA DE HARDWARE MINIMALISTA ARDUINO UNO

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Marco Aurélio Spohn

Este trabalho de conclusão de curso foi defendido e aprovado pela banca avaliadora em: 5/12/2018.

BANCA AVALIADORA

Marco A. Spohn
Prof. Dr. Marco Aurélio Spohn – UFFS

Adriano S. Padilha
Prof. Me. Adriano Sanick Padilha – UFFS

Prof. Dr. Emílio Wuerges – UFFS

**SUMÁRIO**

## AGRADECIMENTOS

À minha mãe, pelo incentivo, amor e apoio, além de me dar tudo e um pouco mais do que precisei durante minha formação.
À minha tia e minha madrinha por estarem sempre dispostas a me ajudar quando precisei.
À minha companheira por ser compreensiva e estar ao meu lado nesses últimos seis meses de academia.
Ao professor Emílio Wuerges por ser mais que um professor, um amigo.
Ao professor e orientador Marco Spohn pela dedicação em me orientar e me guiar nesse ano.

## RESUMO

O presente trabalho apresenta uma análise do código fonte do FreeRTOS *portado* para Arduino Uno. O objetivo dele é dar uma visão mais clara do seu comportamento, além da quantidade de memória que suas estruturas ocupam. Esse trabalho analisa a descrição do livro das funções de criação e manejo de mutexes, semáforos binários, semáforos contadores, timers e funções em caso de uma chamada em uma interrupção. Por fim, traz como resultado o modo como elas se comportam realmente e quanto de memória cada estrutura usada ocupa através da análise de seus respectivos códigos-fonte.

**Palavras-chave**: FreeRTOS. Arduino Uno. *Timer*. *Mutex*. Interrupção.

**ABSTRACT**

This monography presents an analysis about the FreeRTOS source code portable to Arduino Uno. His objective is to give a more accurate and clear vision of his behaviour, besides the quantity of memory usage that his structures uses. This monography analyze the description of the book about the creating functions and management of mutexes, binary and counting semaphores, timers and funcions called inside an interruption. At last, brings as result how they actually behave and how much memory each used structure occupy through an analysis of the respective source codes.
**Keywords**: FreeRTOS. Arduino Uno. Timer. Mutex. Interruption.

# 1 INTRODUÇÃO

Nos dias atuais, os sistemas computacionais estão muito complexos, com vários recursos e componentes. Para um programador escrever um código que se relaciona perfeitamente com esses recursos, é necessário um conhecimento de todo o sistema computacional, o que se torna inviável. Porém, para facilitar a escrita de código dos programadores, existe um software que faz essa interface entre os recursos, os componentes e os códigos. Esse software é o sistema operacional (Tanenbaum; Machado Filho, 1995).

Segundo Tanenbaum; Machado Filho (1995), os sistemas operacionais possuem duas tarefas principais distintas, que são fornecer recursos menos confusos e mais simples e gerenciar esses recursos. Uma tarefa que o sistema operacional realiza, mais especificamente um recurso dele (o escalonador), é a troca de qual aplicação obterá o controle do processador. Em outras palavras, em qual aplicação o processador irá trabalhar. Porém, segundo Barry (2018), algumas aplicações podem possuir requisitos que devem ser tratados em uma determinada janela de tempo. Deste modo, esses programas foram chamados de programas ou aplicações de tempo real.

As aplicações de tempo real geralmente possuem dois tipos de requisitos para funcionar, sendo eles os *soft requirements* e os *hard requirements*. É dito um *soft requirement* quando a sua necessidade máxima de tempo pode ser extrapolada sem comprometer o sistema computacional. Por exemplo, se um *mouse* demorar para responder, será somente irritante, mas ainda será possível usar o sistema. Entretanto, quando é necessário que um evento seja tratado em um certo limite de tempo e as consequências por extrapolar esse limite podem ser catastróficas, essas aplicações são chamadas de *hard requirements* (Barry, 2018). Por exemplo, um sensor de temperatura de uma usina nuclear, um aparelho de hospital que mede os batimentos cardíacos, um controlador de bordo de um avião, etc.

Para ajudar a contornar o problema de tempo, os programas em tempo real podem ter como base um Sistema Operacional de Tempo Real (*Real Time Operating System* – RTOS). Por exemplo o FreeRTOS, o qual é o tema base para esse trabalho, que permite que as aplicações sejam organizadas como várias *threads* de execução e escalonadas (Barry, 2016). O FreeRTOS possui versões para vários dispositivos (Barry, 2018), porém a versão a utilizada não pertence a distribuição oficial, mas a uma versão portável para Arduino Uno (Stevens, 2018).

O Arduino Uno é uma placa eletrônica de design aberto, que em sua essência transforma uma entrada em uma saída, como por exemplo ler de um sensor e enviar essa leitura para outro recurso. A vantagem de usá-lo é que seu custo de aquisição é baixo, é simples de programar e manusear (tem sua própria IDE e linguagem) e tem seu *software* e *hardware open source*. Entretanto, a sua desvantagem é que possui um mi-

crocontrolador limitado, com apenas 32KB de memória *flash*, 2KB de memória SRAM e 1KB de memória EEPROM (AG, 2018). Considerando-se essas limitações, tornou-se necessário abrir mão de várias funcionalidades do FreeRTOS como, por exemplo, algumas opções de gerenciamento de memória, para que houvesse espaço suficiente para a execução da aplicação (Stevens, 2018).

## 1.1 OBJETIVOS

### 1.1.1 Geral

Analisar componentes específicos do *FreeRTOS*, avaliando suas capacidades, suas limitações e seu desempenho, dentro do ambiente do Arduino Uno.

### 1.1.2 Específicos

- Analisar os códigos dos *timers*, das interrupções e do gerenciamento de recursos.
- Analisar as diferenças e mudanças da distribuição oficial do *FreeRTOS* para a versão portável para o Arduino Uno.
- Descrever suas estruturas, definindo os valores que ocupam em memória.
- Examinar sua *performance* e seu comportamento conforme o aumento de *timers*, de recursos e da frequência de interrupções.

## 1.2 JUSTIFICATIVA

Tendo em vista que para iniciar um projeto é necessário ter a compreensão de quais ferramentas usar e que não existe a necessidade de usar instrumentos complexos para resolver problemas pequenos, então ter a noção dos limites dos componentes de *timers*, de interrupções e de gerenciamento de recursos do FreeRTOS no Arduino UNO – uma ferramenta simples – é uma grande vantagem para decidir se essa ferramenta consegue suprir as demandas necessárias da região do projeto em que ela será usada.

# 2 REFERENCIAL TEÓRICO

## 2.1 ARDUINO UNO

O Arduino Uno é uma placa eletrônica de *design* aberto (e.g. se alguma pessoa possuir os mesmos componentes que os que existem na placa original do Arduino, ela pode construí-la). Sua característica principal é o fato dele ser simples e barato, além de possuir sua própria IDE (baseada no *software* Processing) e linguagem (baseada em Wiring) (AG, 2018). Entretanto, suas limitações são um grande contraponto a essa simplicidade. Isso faz com que sistemas operacionais complexos não consigam ser portáveis completamente para o Uno. O trabalho de Stevens (2018) foi esse, ajustar a distribuição do FreeRTOS original, para que seja portável para microprocessadores AVR. Sendo assim, muitos dos recursos foram descartados durante esse processo, como por exemplo a múltipla escolha do modo de armazenamento, que de cinco opções em sua versão original existe apenas uma na versão para o Uno.

A IDE do Arduino, chamada de Arduino Web Editor, que faz parte da plataforma Create, permite ao programador escrever códigos e salvá-los na nuvem, além da IDE sempre estar atualizada, pois é um recurso *online* que somente precisa de um *plugin* para funcionar. A utilidade desse editor está, dentre outras, na facilidade de importar e encontrar bibliotecas, de ter um monitor serial para observar o que está sendo enviado do Arduino para as portas seriais e de poder compilar o código e enviar ao Arduino com um simples clique do *mouse* (AG, 2018).

## 2.2 FREERTOS

Segundo Stankovic; Rajkumar (2004), sistemas operacionais em tempo real tem como diferença de um sistema operacional usual o determinismo e a previsibilidade. Barry (2016) afirma que o FreeRTOS é um *kernel* (escalonador) em tempo real, no qual as aplicações podem ser construídas tendo ele como base, permitindo assim suprir seus requisitos. Ele explica que aplicações são abstraídas como *threads* de execução, comumente chamadas de tarefas. Por fim, declara que o FreeRTOS irá escalonar as tarefas por ordem de prioridades, essas que serão atribuídas pelo escritor do código. Um exemplo é definir baixas prioridades para tarefas menos importantes e altas prioridades para tarefas muito importantes. Assim, o escalonador irá majoritariamente escolher as tarefas mais importantes para serem alocadas na CPU.

Além da funcionalidade de escalonamento, o FreeRTOS possui vários outros recursos úteis para o escritor, tais como serviço de *timers*, armazenamento e interrupções. Isso permite que a escrita do código se torne mais simples, uma vez que estruturas importantes já estarão prontas para uso (Barry, 2016).

O FreeRTOS está atualmente na versão 10.1.1-1, entretanto a versão estudada pelo presente trabalho é a versão 8.2.3. Algumas diferenças da versão 10.1.1-1 para a versão 8.2.3 são que enquanto a versão mais velha somente permite a alocação de memória dinamicamente, as novas versões (9.0.0 em diante) permitem alocação de memória estática em tempo de compilação. Outra diferença é que a partir da versão 9.0.0 é possível forçar uma outra tarefa a ir para o estado bloqueado, enquanto que na versão mais velha não (Barry, 2018).

### 2.2.1 Recursos

A presente seção tem como objetivo explicar alguns recursos disponíveis para uso do FreeRTOS, tendo como principal embasamento o livro de Barry (2016).

#### 2.2.1.1 Tarefas

As tarefas nada mais são que funções escritas na linguagem C. Entretanto, ao contrário das funções, as tarefas nunca devem retornar, mas sim executar em um *loop* infinito e, quando não mais necessárias, devem chamar uma função que se encarregará de deletá-la (isto é, desocupar o espaço ocupado pela alocação dessa tarefa). O autor explica que as tarefas são em sua essência protótipos, podendo assim serem criadas várias tarefas com apenas um protótipo. Toda tarefa, ao ser criada, terá um Task Control Block(TCB) associado a ela. O TCB tem como objetivo guardar as informações de uma tarefa, tais como em que estado ela está, sua prioridade, etc. Quando ela é criada, possuirá sua própria pilha e suas próprias variáveis locais. Em outras palavras, duas tarefas podem ser do mesmo protótipo, no entanto cada uma terá locais de armazenamento de variáveis separadas (Barry, 2016).

Cada tarefa estará em ao menos um estado. Esse estado definirá como o FreeRTOS manejará ela. Os estados possíveis, segundo Barry (2016), são:

- Bloqueado: Quando uma tarefa está no estado bloqueado é porque ela está esperando por algum evento. Esse evento pode ser ou esperar alguma quantidade de tempo ou um evento de outra tarefa, como um dado processado que deve ser enviado a ela, por exemplo. A tarefa também pode bloquear por ambos os motivos. Por exemplo, manter-se bloqueada até um dado chegar ou se passarem 2 segundos, o que acontecer primeiro.

- Suspenso: Tarefas que estão suspensas não podem ser escalonadas. Para isso elas devem chamar uma função que a colocará em estado suspenso. Ela somente sairá desse estado caso outra tarefa a acorde, através de uma outra chamada de função.

- Pronto: Tarefas que estão prontas podem ser escalonadas. Elas não estão sendo processadas, pois existe no momento outra tarefa de prioridade maior ou igual no processador.

- Executando: Quando uma tarefa está nesse estado, significa que ela foi escalonada e está sendo processada na CPU.

As tarefas também possuem uma prioridade associada a elas, podendo esta ser alterada por ela mesma. A prioridade é usada pelo escalonador para decidir qual será a tarefa a ser escalonada. Caso exista mais de uma tarefa com a mesma prioridade, o escalonador escalonará ambas alternadamente (Barry, 2018).

Segundo Barry (2016), sempre deve existir uma tarefa executando. Portanto, uma tarefa com a menor prioridade possível é criada ao iniciar o escalonador: a tarefa *idle*. O fato de sua prioridade ser a menor, faz com que ela somente seja executada quando nenhuma outra tarefa está sendo processada. Ele afirma que caso alguma tarefa com prioridade maior entre no estado pronto, essa nova tarefa imediatamente será escalonada e entrará no lugar da *idle*.

A tarefa *idle* pode ter funções dentro dela, chamadas de *hook*, que serão chamadas a cada iteração do *loop* da tarefa. Esse *hook* é geralmente usado, mas não se limita a, executar funcionalidades que não possuem muita prioridade. Entretanto, é necessário que as funções de *hook* não demorem muito para retornar para a tarefa *idle*, pois é ela que realiza a limpeza do *kernel* com os dados que devem ser liberados (Barry, 2016). Então, se houver demora pode faltarem dados para novas tarefas que serão criadas, por exemplo.

#### 2.2.1.2 Escalonador

Os sistemas operacionais permitem que múltiplas aplicações sejam executadas simultaneamente, entretanto não é possível que mais de uma tarefa seja alocada por vez à CPU. É papel do escalonador realizar e escolher qual aplicação será alocada, tendo como base um dos seguintes algoritmos de escalonamento (Tanenbaum; Machado Filho, 1995):

- Escalonamento priorizado, preemptivo com fatias de tempo: Algoritmo de escalonamento que é geralmente encontrado nos sistemas operacionais em tempo real. Possui prioridade fixa pois o algoritmo não muda a prioridade de uma tarefa, mas não previne que alguma tarefa mude sua própria prioridade ou a de outra. É preemptivo porque involuntariamente move a tarefa do estado de executando para o estado de pronto para permitir que outra tarefa execute. Com fatias de tempo devido ao fato dele usar uma quantidade de tempo igual para

processar cada tarefa. Esses algoritmos irão escolher uma nova tarefa para entrar no estado executando quando o tempo dado a tarefa que está no processador neste momento acabar (Barry, 2016).

- Escalonamento priorizado, preemptivo sem fatias de tempo: Utiliza quase o mesmo sistema que o escalonamento com fatias de tempo, a diferença é que sem fatias de tempo, só haverá a troca de contexto se uma tarefa de maior prioridade entrar no estado pronto ou a tarefa que está executando entrar no estado bloqueado. Assim, haverá menos trocas de contexto e diminuirá a carga de processamento do escalonador, entretanto poderá aumentar a discrepância entre a quantidade de tempo que cada tarefa ficará executando (Barry, 2016).
- Escalonamento cooperativo: Nesse escalonamento, as tarefas irão executar até que explicitamente liberem o uso do processador ou entrem no estado bloqueado (Barry, 2016).

#### 2.2.1.3 Gerenciamento de memória

A versão 8.2.3 do FreeRTOS somente permite que os objetos do *kernel* sejam alocados dinamicamente. Entretanto, a partir da versão 9.0.0, esses objetos podem ser alocados estaticamente em tempo de compilação. Isto é, sempre que um novo objeto tal como uma fila, uma tarefa ou um *mutex*, por exemplo, tenha que ser criado, essa ação será feita dinamicamente (Barry, 2016).

A distribuição original do FreeRTOS permite que seja usado um dos cinco modos de alocação de memória (Barry, 2016). Contudo, na versão proposta por Stevens (2018), somente a *heap* 3 permaneceu. Essa opção usa as funções padrões *malloc* e *free* para alocar e liberar memória (Barry, 2016).

#### 2.2.1.4 *Timers*

*Timers* são usados ou para agendar a execução de uma determinada ação ou para repetir uma determinada ação a cada intervalo de tempo. Eles estão sob o controle do *kernel* e não possuem relação com os *timers* de *hardware*. *Timers* que somente executam uma ação apenas uma vez são chamados de *timers one-shot* e podem ser reiniciados manualmente. *Timers* que executam a cada período de tempo são chamados de *timers auto-reload* e a cada vez que terminam a ação programada, se reiniciam (Barry, 2016).

Existem dois estados possíveis para os *timers*, o estado *dormant* (que é quando ele existe e pode ser referenciado, mas que não está executando e portanto não irá disparar em momento algum) e o estado *running* (ele disparará quando o seu tempo esgotar e avisará o *kernel* de que acabou, onde será realizada uma ação pré-estabelecida) (Barry, 2016).

#### 2.2.1.5 Interrupções

Segundo Barry (2016), geralmente é necessário que alguma ação seja feita caso um evento ocorra, sendo a funcionalidade que capta esses eventos chamada de interrupção. Um exemplo disso é se o sensor perceber que a temperatura de uma usina nuclear está muito alta, ele deve então avisar ao sistema de que é necessário abaixar a temperatura de algum modo. O sistema irá receber essa interrupção e irá usar os modos cabíveis a ele para abaixar a temperatura para um nível aceitável. Outro exemplo de interrupção, é quando um usuário aperta qualquer tecla de um teclado, que gerará uma interrupção e será tratada de acordo com qual tecla foi pressionada e em que aplicação foi apertada essa tecla.

Ele continua, afirmando que em sistemas reais, as aplicações devem responder a vários eventos (e consequentemente, a várias interrupções), que possuem uma necessidade diferente tanto de processamento como de uso de recursos. O FreeRTOS não limita como esse processamento deve ocorrer (entretanto alerta ao escritor para manter o tratamento da interrupção o mais rápido possível, deixando para uma outra tarefa cuidar do que deve ser feito com essa interrupção), mas implementa funcionalidades que ajudam de maneira geral a tratar essas interrupções, tornando a escrita do código mais simples.

As interrupções também possuem uma prioridade, todavia com grandes diferenças em relação a prioridade dada a uma tarefa. A prioridade de uma tarefa sempre será menor que a prioridade de uma interrupção, ou seja, a tarefa de maior prioridade ainda assim será deixada de lado caso uma interrupção com a menor prioridade possível ocorra. Isso se deve ao fato de que uma interrupção é controlada pelo *hardware*, enquanto que uma tarefa é controlada pelo *software*. Como é o *hardware* que irá dizer como e quando essa interrupção será executada, a prioridade de escalonamento deve ser a interrupção (Barry, 2016).

#### 2.2.1.6 Gerenciamento de recursos

Segundo Barry (2016), como o sistema operacional permite que mais de uma tarefa execute ao mesmo tempo, podem ocorrer alguns problemas, tais como o de uma tarefa acessar um recurso que está sendo usado por outra, corrupção de dados, etc. Isso ocorre pois não é necessário que uma tarefa libere seus recursos ao ser removida da CPU quando ocorre uma troca de contexto. Então, se por exemplo, existe uma tarefa usando a impressora e ela perde a CPU para outra tarefa que também possui o controle da impressora, a impressão resultante provavelmente será uma mistura das as duas tarefas, e não de cada uma separada, que é o certo a ocorrer.

O FreeRTOS possui métodos para evitar a ocorrência desses problemas. Um de-

les é a região crítica, essa região somente permitirá que uma tarefa a acesse por vez, isso significa que nunca duas ou mais tarefas entrarão nessa região de código. Esse fato é possível pois todas as interrupções (ou interrupções com prioridades menores que uma estabelecida) são desabilitadas ao entrar em uma região crítica. Como o escalonador somente executará a troca de contexto quando ocorre a interrupção que sinaliza a necessidade de realizar a troca de contexto, essa tarefa continuará executando até que saia da região crítica (Barry, 2016).

Para evitar que um recurso seja utilizado por mais de uma tarefa, o FreeRTOS proporciona um modo que permitirá que somente uma tarefa tenha controle do recurso por vez, que consiste em variáveis de controle, chamadas de *mutexes*. Eles podem ser pensados como a chave de um recurso, no qual somente a tarefa que a possuir poderá acessar o recurso (Barry, 2016).

Por fim, um recurso pode ter uma tarefa *gatekeeper*, que é uma tarefa que possuirá o controle total do recurso cem por cento do tempo. Isso significa que somente ela poderá acessar ou dar ordem ao recurso associado a ela (Barry, 2016). Um exemplo disso é uma tarefa *gatekeeper* para a impressora, que armazenará as requisições por alguma ordem, escolhendo um texto por vez e imprimindo-o.

# 3 TRABALHOS RELACIONADOS

Ao proceder à revisão bibliográfica, foram utilizados os mecanismos de busca *IEEE* e *Google Scholar*. As *strings* de busca usadas foram *"FreeRTOS on Arduino Uno"*, *"FreeRTOS Arduino Uno"* e *FreeRTOS*. Entretanto, não foram encontrados trabalhos que abordassem uma análise da *performance* dos componentes do FreeRTOS no Arduino Uno. Ademais, trabalhos moderadamente relacionados foram encontrados:

- O trabalho de Buonocunto et al.(2016) utiliza um RTOS no Arduino Uno para garantir suporte a *multitasking*.
- O trabalho de Sacha(1995) discute métodos de avaliar funcionalidades de sistemas operacionais em tempo real, tais como sincronização de tarefa, gerenciamento de mensagens e gerenciamento de eventos. Ademais, os autores ainda sugerem alguns exemplos de métricas a serem usadas para fazer essa avaliação.
- O trabalho de Pereira et al.(2014) implementa o FreeRTOS, entretanto fazem de um modo que desembargam parte dos serviços de tempo real para uma placa FPGA. Para a análise, utilizam métricas como latência e previsibilidade.

# 4 METODOLOGIA

No presente trabalho foi analisado os códigos da biblioteca FreeRTOS, portada para Arduino Uno. Essa biblioteca pode ser encontrada e instalada através da Arduino IDE. A versão analisada e testada é a versão 8.2.3.

Como base para a análise foi usado o livro *Mastering the FreeRTOS Real Time Kernel – A Hands-On Tutorial Guide*, escrito por Richard Barry. Cada capítulo abordado neste trabalho foi antes lido no livro, sintetizado e em seguida comparado e analisado com sua respectiva função ou macro no código.

Como cada capítulo do livro é, na maioria dos casos, referente a um recurso, houve a necessidade de além de analisar o comportamento dele, também verificar as estruturas que são usadas durante a execução do código (tais como a quantidade de memória que ocupam ao serem alocadas e inicializadas, qual o intuito dessa variável para a estrutura, etc).

A ferramenta utilizada para realizar a análise do código e confirmar sua execução foi um Arduino Uno. Nele foram realizados testes para confirmar os tamanhos de cada estrutura. Os testes consistiram em:

- Ao chamar a função que cria uma fila, ela retornasse o tamanho de uma estrutura de fila (*queue_t*).
- Ao chamar a função de criar um *timer*, retornar o tamanho da estrutura *timer_t*.
- Imprimir os tamanhos de *ListItem_t*, *List_t* e *MiniListItem_t*.

O algoritmo usado para realizar os testes pode ser encontrado nos anexos e os resultados podem ser vistos no capítulo a seguir.

# 5 RESULTADOS E ANÁLISE

## 5.1 ESTRUTURAS PRIMITIVAS

Estudando a quantidade de *bytes* que alguns tipos de dados ocupam na memória do FreeRTOS, os tipos básicos são:

- *uint8_t*: Ocupa 1 *byte*. De acordo com a ISO (1999), é um *unsigned int*, que conterá exatamente 8 *bits*. O mesmo pode ser percebido para tipos *uint16_t* e *uint32_t*, que são respectivamente exatamente 16 e 32 *bits* (logo, 2 e 4 *bytes*). As definições de tipos que contém como início **u**, seja a letra em minúsculo ou maiúsculo, significa que as variáveis não possuem sinal, e.g. são sempre positivas.
- *port_STACK_TYPE*: É definido como um *uint8_t* e portanto, ocupa 1 *byte*.
- *StackType_t*: É definido como um *port_STACK_TYPE* e portanto, ocupa 1 *byte*.
- *BaseType_t*: É definido como um *signed char* e portanto, ocupa *1 byte*.
- *UBaseType_t*: É definido como um *unsigned char* e portanto, ocupa 1 *byte*.
- *TickType_t*: Pode ser definido de duas maneiras, dependendo da definição de configuração *configUSE_16_BIT_TICKS*. Em arquiteturas onde *configUSE_16_BIT_TICKS está definido como 1*, é um *uint16_t*, em arquiteturas onde *configUSE_16_BIT_TICKS* está definido como 0, é um *uint32_t*. Observando o arquivo *FreeRTOSConfig.h* percebe-se que na versão do FreeRTOS *portada* para Arduino Uno possui a definição *configUSE_16_BIT_TICKS* como 1, então esse tipo é dado como um *uint16_t* e consequentemente ocupa 2 *bytes*.

## 5.2 SEMÁFOROS

Os semáforos são divididos em três categorias: os *mutexes*, os *mutexes* recursivos e os semáforos binários.

### 5.2.1 *Mutexes*

Analisando a definição da estrutura dos *mutexes* (definida em *semphr.h*), percebe-se que este tipo ocupa (para cada variável dele declarada) 31 *bytes*. Entretanto, devido a alocação de memória do *malloc*, é utilizado mais dois *bytes* de controle. Assim, esse tipo de dado ocupa no total, após ser criado (alocado e inicializado), 33 *bytes*. A estrutura completa, com a quantidade de *bytes* correspondente a cada variável declarada dentro da estrutura pode ser vista na seção Anexos.

#### 5.2.1.1 Definição e Variáveis

*Mutexes* são em sua essência filas, tanto que suas funções são definições de outras funções genéricas que as criam e as alteram. Entretanto, dentro dessas funções genéricas haverá um comportamento diferente em algumas regiões de código se a fila passada por parâmetro é um *mutex*. O algoritmo 1 é uma demonstração desse comportamento. Esse algoritmo verifica se a fila é um *mutex* e, em caso positivo, libera a posse dele da tarefa que o previamente adquiriu.

Ao contrário das filas, os *mutexes* não possuem nenhum dado salvo na área de armazenamento, apenas os metadados são usados e tem como intuito gerenciar qual tarefa possui controle do recurso guardado por esse *mutex*.

Algoritmo 1 – Exemplo de comportamento especial de uma fila *mutex*

```
1 if pxQueue->uxQueueType == queueQUEUE_IS_MUTEX then
2 |   xYieldRequired = xTaskPriorityDisinherit( ( void * ) pxQueue->pxMutexHolder );
3 |   pxQueue->pxMutexHolder = NULL;
4 end
```

As variáveis de uma fila normal que são interessantes para o contexto de *mutex* são:

- *PcHead*: Variável que indica o início da área de armazenamento. Como *mutexes* não precisam armazenar dados, essa variável é usada para indicar que a fila é um *mutex*. Basta atribuir *NULL* a ela.

- *PcTail*: Aponta para o fim da área de armazenamento da fila. No caso de *mutexes*, essa variável é chamada (através de um *typedef*) de *pxMutexHolder*, pois tem o intuito de guardar qual tarefa tem o controle daquele *mutex*.

- *xTasksWaitingToSend* e *xTasksWaitingToReceive*: São listas de tarefas que desejam colocar ou remover um dado na fila, respectivamente. No contexto de *mutexes*, serão listas de tarefas querendo adquiri-lo para utilizar o recurso guardado por ele ou liberar o controle exclusivo do recurso após utilizá-lo.

- *uxMessagesWaiting*: Variável que conta quantas mensagens estão na fila. Incrementa quando uma tarefa escreve um dado na fila e decrementa quando uma tarefa remove um dado da fila. Como apenas uma tarefa pode acessar um recurso por vez, então *uxMessagesWaiting* será sempre alternada entre 0 e 1.

- *uxLenght*: Utilizada para armazenar a quantidade máxima de dados que determinada fila irá armazenar. No contexto de *mutexes*, como apenas uma tarefa pode acessar um recurso por vez, o tamanho máximo é 1.

#### 5.2.1.2 Inicialização

Antes de utilizar um *mutex*, é necessário chamar uma função que criará e inicializará todas as estruturas e variáveis dele (chamada de *xSemaphoreCreateMutex*, contudo é apenas uma definição para *xQueueCreateMutex*). Essa função terá como retorno um *handle*[1] e é através desse *handle* que as funções de adquirir ou liberar o controle do recurso protegido saberão em qual *mutex* eles devem agir.

A função de inicialização irá alocar a quantidade de memória necessária para a estrutura e em seguida salvará a informação de que a fila é um *mutex* e que nenhuma tarefa tem controle dele. Então inicializará as listas de tarefas que estão aguardando para escrever ou ler da fila.

Sabendo que quando uma tarefa desejar ter controle do recurso correspondente a esse *mutex*, ela pegará um item dessa fila e quando desejar liberar o controle do recurso, devolverá o item, então é necessário atribuir 1 à variável que controla quantos itens a fila possui. Assim, quando o *mutex* for pego pela primeira vez, a quantidade de itens na fila estará em zero, significando que o *mutex* já foi pego e evitará que outras tarefas consigam fazer o mesmo.

Por fim, essa função retorna um *handle* para a tarefa que o chamou.

#### 5.2.1.3 Adquirir o Controle do Recurso

Com as estruturas inicializadas, os *mutexes* já podem ser adquiridos e seus recursos – se inicializados também – já podem ser utilizados. Para tal, antes de utilizar o recurso é necessário entrar na região crítica[2], adquirir o *mutex* e executar o trecho protegido por ele.

A função que faz a tarefa adquirir o *mutex* é a *xSemaphoreTake* (contudo é apenas uma definição para *xQueueGenericReceive*), que tem como parâmetros o *mutex* que deseja pegar e o tempo de espera. O último é usado em casos onde o *mutex* já foi pego. Deste modo, a tarefa aguardará, durante esse tempo de espera, o *mutex* ser devolvido. Caso esse tempo extrapole e a tarefa ainda não adquiriu ele, será alertado um erro.

A função verificará se alguma tarefa já possui controle do *mutex*. Em caso positivo, isto é, caso o *mutex* já foi pego, não haverá nenhuma mensagem.

Se houver uma mensagem esperando para ser lida, o *mutex* está disponível. Então será diminuida a quantidade de mensagens aguardando (significa assim que não

[1] Handles são ponteiros para *void* de 2 *bytes* cada, usados para indicar em qual objeto deseja fazer alterações.

[2] São regiões onde a atomicidade deve ser preservada, e.g. somente uma tarefa pode estar executando esse trecho de código. Outras tarefas que desejarem entrar nessa região de código deverão esperar a primeira sair dela. As funções que entram e saem da região crítica são respectivamente *taskENTER_CRITICAL e taskEXIT_CRITICAL*.

existe nenhuma e consequentemente que o *mutex* já foi adquirido por uma tarefa) e será guardado a tarefa atual como a que possui controle desse recurso.

Existe a possibilidade do *mutex* já ter sido pego por alguma outra tarefa, por isso o FreeRTOS implementa um tempo de espera que fará com que ela aguarde um certo tempo definido enquanto continua tentando pegar o *mutex*. Deste modo, irá ocorrer um dos seguintes casos:

- Se o tempo de espera for 0: Significa que a tarefa não deve esperar tempo algum (ou ela pega o *mutex* ou não consegue e desiste de pegá-lo). Assim, a função retorna que houve erro ao tentar pegar o *mutex*, pois a fila estava vazia.

- Se o tempo de entrada ainda não foi atribuído: Então ele irá definir o tempo em que essa chamada entrou nesse caso pela primeira vez (para que seja possível verificar se o tempo limite expirou).

Após fazer todos os procedimentos acima, a tarefa sairá da região crítica. Entretanto, o escalonador será suspenso[3], a fila bloqueada para garantir a atomicidade e por fim será verificado o tempo de espera.

Se o tempo de espera ainda não expirou, se existe algum dado na fila e se a fila é um *mutex*, então haverá a herança de prioridade[4]. A seguir, o TCB será atribuido à lista de mensagens que estão esperando para ler um dado do *mutex*. Como a função de ter posse do *mutex* é um *loop* infinito, na próxima iteração a tarefa terá o controle do recurso e poderá continuar executando o que foi programada.

Caso o tempo de espera não tenha sido expirado ainda mas o *mutex* já tenha sido pego por outra tarefa, então a função tentará de novo na próxima iteração do *loop* infinito.

Se o tempo de espera extrapolou, então será retornado erro à tarefa que o chamou, esse erro significa que o *mutex* já estava em posse de outra tarefa e não foi possível adquiri-lo antes que seu tempo de espera tenha sido expirado. No fim de todos esses casos a fila será desbloqueada e o escalonador retomado.

#### 5.2.1.4 Liberar Controle do *Mutex*

Para devolver o *mutex* e assim permitir que outra tarefa tenha controle do recurso, basta chamar a função *xSemaphoreGive* (que é uma definição para *xQueueGive*), tendo como parâmetros o *mutex* e o tempo de espera.

[3] Isso permite que as interrupções funcionem normalmente, contudo não haverá troca de contexto.

[4] Se a tarefa atual tem maior prioridade que a tarefa que possui o *mutex*, então a segunda terá sua prioridade aumentada para a prioridade da primeira. Essa técnica é usada para que a tarefa que tem maior prioridade execute o seu procedimento o mais rápido possível, uma vez que permite à outra liberar o *mutex* mais cedo.

Inicialmente, a função entrará em uma região crítica e então verificará se a quantidade de elementos na fila é menor que a capacidade máxima de elementos que ela consegue armazenar (no contexto de *mutexes*, isso sempre será verdade). Se a fila é um *mutex*, então será chamada uma função para deserdar a prioridade da tarefa que o chamou (essa execução sempre acontece pois as funções para herdar e deserdar são genéricas, sendo feito apenas uma comparação de prioridade para herdar ou não). Depois atribuirá *NULL* à variável que controla qual tarefa tem o controle do *mutex*. Em outras palavras, deserdará a tarefa atual e simbolizará que nenhuma tarefa atualmente possui controle daquele recurso.

Para verificar se é possível ou não uma tarefa possuir o *mutex*, é analisado a quantidade de mensagens na fila. Logo, ao devolver o *mutex*, é necessário aumentar essa quantidade, para que assim quando outra tarefa for verificar, confirmar que o *mutex* pode ser pego.

Sabendo agora que o *mutex* foi devolvido, é verificado se não existe alguma tarefa esperando para ter posse daquele *mutex* e, em caso positivo, essa tarefa será imediatamente desbloqueada[5].

Por fim, seguindo a execução de código, a tarefa sairá da região crítica e retornará *pdPASS* (o que significa que ela conseguiu devolver o *mutex*).

### 5.2.2 *Mutexes* Recursivos

Existe a possibilidade de uma tarefa precisar pegar o mesmo *mutex* mais de uma vez, para isso existem os *mutexes* recursivos. Eles são em sua estrutura e execução bastante similares aos *mutexes* normais. A sua diferença está no fato deste tipo de *mutex* guardar quantas vezes ele foi pego. Assim, uma tarefa pode pegar ele quantas vezes quiser e só realmente o devolverá quando o número de vezes que o *mutex* foi devolvido for igual ao numero de vezes que ele foi pego.

Na prática, isso significa que quando a tarefa pegar um *mutex* recursivo (através da função *xSemaphoreTakeRecursive* – encrontrada em *semphr.h* – sendo ela uma definição para *xQueueTakeMutexRecursive*), irá incrementar o contador de vezes que aquele *mutex* foi pego. Caso seja a primeira vez que a tarefa está adquirindo o *mutex*, então também será atribuído à fila correspondende ao *mutex* o TCB de qual tarefa o adquiriu. Ao devolver o *mutex* (através da função *xSemaphoreGiveRecursive* – encontrada em *semphr.h* – que é uma definição para *xQueueGiveMutexRecursive*), o contador é decrementado em um. Caso esse contador chegue a 0, então será desatribuído à fila do *mutex* correspondente o TCB da tarefa que chamou a função.

[5] Se a tarefa desbloqueada possuir maior prioridade, então a tarefa atual deverá conceder o uso do processador imediatamente.

### 5.2.3 *Counting Semaphores*

*Counting semaphores* são semáforos que podem ser pegos e devolvidos mais de uma vez. Esses semáforos podem ser usados para dois propósitos. O primeiro, com o propósito de contar quantos eventos aconteceram. Assim, sempre que uma tarefa devolver um semáforo (pela função padrão *xSemaphoreGive*) significa que aconteceu um evento. A segunda é usar o semáforo como um contador de quantos recursos existem disponíveis no momento. Toda vez que um recurso é pego (através da função padrão *xSemaphoreTake*), é diminuido em um o contador de semáforo (Barry,2016).

O semáforo é implementado de forma que sua fila possa conter mais de um item. Assim, ao chamar a função de criar um *counting semaphore* (através da função *xSemaphoreCreateCounting*, encontrada em *semphr.h*) é passado como parâmetro o número máximo que ele contará e o contador inicial daquele semáforo. Se ele for usado para o primeiro caso, então inicia-se esse contador como 0. Se for usado para o segundo, inicia-se com a quantidade máxima de recursos disponíveis.

### 5.2.4 Semáforos Binários

Semáforos binários são quase iguais aos *mutexes*. Sua única diferença é que semáforos binários não utilizam a técnica de herança de prioridade. Para criar um semáforo binário basta chamar a função *vSemaphoreCreateBinary* (encontrada em *semphr.h*). Desconsiderando a herança de prioridade, sua inicialização e *modus operandi* é igual a de um *mutex*.

## 5.3 *TIMERS*

Os *timers* tem uma estrutura própria, definida em *timers.c*. Analisando essa estrutura, percebe-se que ela ocupa 19 *bytes*, mas como é construída através de um *malloc*, ocupará dois a mais. Assim, um *timer* alocado e inicializado, ocupa 21 *bytes*. A estrutura completa com o tamanho de cada variável pode ser encontrada na seção de anexos.

### 5.3.1 Inicialização

Antes de criar um *timer*, é necessário criar uma tarefa que gerenciará todos eles. Essa tarefa, chamada de tarefa *daemon*, é alocada e inicializada automaticamente ao chamar a função que inicia o escalonador (*vTaskStartScheduler*, encontrada em *tasks.c*). Como o *daemon* é uma tarefa, então o custo de alocar ela será de 130 *bytes* (se usadas as configurações padrão da biblioteca, quando instalada pela IDE do Arduino. Sendo

41 do TCB, 4 dos *mallocs* – 2 do TCB e 2 da tarefa – e *configMINIMAL_STACK_SIZE*[6] de pilha, que tem como quantidade padrão 85 *bytes*).

### 5.3.2 Estrutura

A estrutura de um *timer* consiste em:

- *pcTimerName*: Ponteiro que aponta para uma *string*. É o nome do *timer*, usada geralmente para *debug*.
- *xTimerListItem*: Uma lista encadeada padrão, usada pelo *kernel* para gerenciar eventos.
- *xTimerPeriodInTicks*: O período do *timer*. Usado para atualizar o seu tempo restante, reiniciar e iniciar *timers*, etc.
- *uxAutoReload*: Controla se o *timer* é um *timer auto-reload* ou não.
- *pvTimerID*: Ponteiro para *void* que identifica ele. É útil quando existem vários *timers* que tem a mesma função de ativação.
- *pxCallbackFunction*: Função que será chamada quando o *timer* ativar.

Para criar um *timer* é usada a função *xTimerCreate* (encontrada em *timers.c*). Essa função irá primeiro verificar se o período não é 0 (nenhum *timer* pode ter período 0), irá alocar o tamanho necessário. A seguir, os detalhes dele serão atribuídos às suas variáveis, tais como o nome, o período, se é um *timer auto-reload*, etc. Por fim, retornará o *handle* do *timer*.

### 5.3.3 A Tarefa *Daemon*

O FreeRTOS possui uma lista de *timers* ativos (funcionalmente são duas, a lista ativa e a lista de *overflow*, que se alternam à medida que o contador de tempo do FreeRTOS sofre *overflow*), por ordem de qual expirará primeiro. Deste modo, o trabalho da tarefa *daemon* será primeiro obter o tempo do *timer* mais próximo a ativar. Se o tempo sofreu *overflow*, então será feita uma troca das listas. Essa troca consiste em verificar se a lista atual (antes de ser trocada) está vazia. Se não estiver, então serão processados todos os *timers* dessa fila (o modo de processamento é chamar a função que deve ser executada quando o *timer* expirar), atualizados os tempos dos *timers auto-reload* e serão inserindos de novo na lista de *timers* ativos, com seus tempos atualizados.

Caso a fila de *timers* ativos esteja vazia, isto é, não existe nenhum *timer* ativo no momento, o valor da variável que guarda qual é o mais próximo de expirar será

[6] Variável encontrada em *FreeRTOSConfig.h*.

definido para 0 (como se fosse simulado um *timer* expirando no tempo 0 da outra fila). Assim, quando acontecer o *overflow* e as listas forem trocadas, o tempo atual será maior ou igual a 0 e o *daemon* verificará novamente qual o tempo mais próximo a expirar na outra lista.

Tendo o tempo, será chamada a função *pvProcessTimerOrBlockTask* (encontrada em *timers.c*). Essa função obterá o tempo atual e verificará se a lista de *timers* foi trocada.

Caso tenha acontecido a troca de listas, não é necessário fazer nada, uma vez que ao trocá-las, todos os *timers* que estavam na lista antiga terão sido processados pela função que realiza essa troca. Entretanto, caso elas não tenham sido trocadas, é verificado se existe algum *timer* na lista e se o próximo tempo para expirar já passou. Em caso positivo, deve-se retomar o escalonador e executar a função chamada pelo *timer*.

Entretanto, quando o próximo tempo ainda não foi atingido, a tarefa *daemon* verificará se ambas as listas de *timers* estão vazias. Em caso positivo ela irá suspender indefinidamente, até que um comando seja recebido (pois se não existe nenhum *timer* ativo, não há a necessidade da tarefa *daemon* ser escalonada, uma vez que somente ficaria ociosa aguardando um comando). Caso exista ao menos um *timer* em uma das listas, então a tarefa será colocada na lista de atrasados ou até que o tempo do *timer* mais próximo expire, ou até que um comando seja recebido. Logo depois de um desses dois casos acontecer, o escalonador será retomado e, se houver uma tarefa que deva tomar posse do processador (devido à prioridade), a tarefa *daemon* irá cedê-lo.

Caso seja realizado um comando ou um *timer* expire, a tarefa *daemon* será imediatamente desbloqueada (no segundo caso a tarefa também será desbloqueada pois o tempo máximo de bloqueio é o tempo até o próximo *timer* expirar). Assim, será chamada a função *prvProcessReceivedCommands* (encontrada em *timers.c*), que irá entrar em *loop* enquanto houver algum comando na fila de comandos de *timers* (se a tarefa *daemon* desbloqueou devido a um *timer* expirado, é nessa condição de laço que a função será pulada). Dentro desse *loop*, será primeiro adquirido qual é o *timer* referido e a seguir, caso ele esteja em alguma lista, será removido dela. Por fim, será adquirido o tempo atual e o comando será processado. Os comandos podem ser:

- Iniciar ou reiniciar: O comportamento será o mesmo (pois se ele estivesse ativo, foi removido e não estará mais). Esse comportamento consiste em chamar a função *prvInsertTimerInActiveList* (encontrada em *timers.c*), que irá inserir o *timer* na lista de *timers* ativos, tendo dois valores possíveis como retorno. Se o valor for *pdTRUE*, então significa que o *timer* já expirou e deve ser processado instantaneamente (sem inserir na fila de *timers* ativos). No caso do *timer* ser um *timer auto-reload*, então após ser processado, deve ser inserido na fila de comandos um

comando de iniciar ele novamente. Se o retorno for *pdFALSE*, significa que ele foi inserido na lista normalmente.

- Parar: Esse comando já foi processado ao retirá-lo da fila de *timers* ativos, deste modo não é necessário fazer mais nada.
- Alterar Período: É necessário apenas inseri-lo novamente na fila de *timers* ativos, entretanto com o novo período. O retorno esperado é sempre *pdFALSE*. Como não se sabe quanto tempo faltava para expirar, o tempo usado como referência é o tempo atual. Assim, sabendo que o período de um *timer* não pode ser 0, o tempo em que ele irá expirar deve estar obrigatoriamente no futuro.
- Deletar: Sabendo que antes de ser verificado qual comando é, o *timer* foi removido da lista de *timers* ativos, o que resta é apenas liberar a memória que ele usava. Assim, apenas chama-se a função *vPortFree* (encontrada em *heap3.c*), que irá realizar esse trabalho.

Assim, o *loop* principal da tarefa *daemon* consiste em:

- Adquira o tempo do *timer* mais próximo a ativar;
- Suspenda o escalonador, altere as listas e processe os *timers* se necessário. Depois aguarde um *timer* expirar ou um comando chegar, suspendendo caso não exista nenhum ativo ou atrasando até o tempo do próximo.
- Processe os comandos;

## 5.4 INTERRUPÇÕES

Quando uma interrupção ocorre, ela deve ser tratada. Para tratar essas interrupções, geralmente serão usadas funções do FreeRTOS. Essas funções não podem bloquear, pois não foi uma tarefa que chamou essa função e logo, não existe uma tarefa para colocar no estado de bloqueado. Assim, foram criadas funções que podem ser chamadas dentro de uma ISR (*Interruption Service Routine*). Entretanto esse não é o único motivo. Interrupções também devem ser tratadas o mais rápido possível, logo essas funções são otimizadas para serem mais rápidas e eficientes.

### 5.4.1 A Variável *xHigherPriorityTaskWoken*

Quando uma tarefa de maior prioridade que a tarefa atual é colocada no estado de pronto, ela deve ser processada imediatamente, do contrário não estaria de acordo com a política de escalonamento preemptivo por prioridade. Contudo, quando essa

troca de contexto ocorre pode variar um pouco, dependendo de onde foi que ocorreu essa chamada de troca.

Quando a troca ocorre dentro de uma função do FreeRTOS, a troca de contexto ocorre imediatamente (através da função *portYIELD_WITHIN_API*, encontrada em *Arduino_FreeRTOS.h*). Já quando ocorre dentro de uma interrupção, pode ser mais vantajoso que a troca seja feita depois, para que toda a interrupção seja processada primeiro (seguindo assim as necessidades de serem tratadas o mais rápido possível). Deste modo, ao invés de realizar a troca de contexto imediatamente, elas alteram essa variável e, quando saírem da interrupção, processam a troca. Entretanto, o escritor do código deve primeiro atribuir essa variável para *pdFALSE*. Ademais, é deixado também a cargo do escritor definir quando realizar a troca de contexto, pois não é feito automaticamente pelo FreeRTOS. (Barry, 2016).

### 5.4.2 Filas e Semáforos

O uso da variável xHigherPriorityTaskWoken é facilmente percebido nas funções *xQueueReceiveFromISR*, *xQueueGiveFromISR* (essa e a primeira encontradas em *queue.c*), *xSemaphoreTakeFromISR* e *xSemaphoreGiveFromISR* (encontradas em *semphr.h*).

#### 5.4.2.1 *xQueueReceiveFromISR*

Na função *xQueueReceiveFromISR* (e consequentemente *xSemaphoreTakeFromISR*), como não é possível bloquear a fila, será apenas verificado se existe um dado na fila e em caso positivo decrementado a quantidade de mensagens.

Se ela estiver desbloqueada, então será verificado se existe alguma tarefa desejando escrever na fila. Em caso positivo, como essa tarefa agora pode escrever (visto que um dado foi retirado), ela será desbloqueada. A função que realiza o desbloqueio retornará *pdTRUE* se a tarefa desbloqueada possuir prioridade maior que a atual. Assim, ao invés de imediatamente liberar o controle do escalonador, a variável *pxHigherPriorityTaskWoken* será alterada para 1 e será retornado *pdPASS*.

Se a fila estiver bloqueada, então será incrementado um contador (*xRxLock*) de quantas tarefas foram removidas dela. Esse contador é usado para quando desbloquear a fila, saber quantos dados foram removidos. A seguir também será retornado *pdPASS*.

Se não houver mensagens na fila, então apenas será retornado *pdFAIL*, pois não é possível esperar um dado ser inserido visto que interrupções devem realizar seu trabalho o mais rápido possível.

#### 5.4.2.2 *xQueueGiveFromISR*

Nessa função (e consequentemente na função *xSemaphoreGiveFromISR*), é primeiro verificado se há espaço na fila para colocar mais um item e depois aumenta-se o número de mensagens na fila sem deserdar a tarefa. Isso se deve ao fato de que a herança de prioridade somente ocorre em *mutexes* e como *mutexes* não podem ser dados ou pegos dentro de interrupções, então não há essa necessidade.

Após aumentar o número de mensagens é verificado se a fila está desbloqueada e se existe alguma tarefa que deseja adquirir o controle daquele semáforo. Em caso positivo, aquela tarefa será desbloqueada e a variável *xHigherPriorityTaskWoken* será alterada para 1.

Se a fila está bloqueada, então é apenas aumentado o contador de tarefas que escreveram naquela fila, para que seja atualizado esse contador quando a fila for desbloqueada.

Se não houver espaço disponível, então será retornado um erro que significa que a fila estava cheia. Nos outros casos será retornado *pdPASS*.

#### 5.4.2.3 *xQueueGenericSendFromISR*

Para essa função, é verificado se a quantidade de mensagens dentro da fila é menor que o tamanho máximo dela ou se é para sobrescrever algum dado na fila. Caso algumas dessas condições forem satisfeitas, então o dado será transferido à fila.

Em seguida, se a fila estiver desbloqueada, é verificado se existe alguma tarefa que está bloqueada esperando para ler da fila e, em caso positivo, a tarefa é colocada no estado pronto (entretanto ainda não é escalonada se possuir prioridade maior que a tarefa atual) e a variável *xHigherPriorityTaskWoken* é alterada para 1.

Caso a fila esteja bloqueada, o contador de itens que foram inseridos na fila enquanto ela estava bloqueada é aumentado em um. Assim, quando a fila desbloquear os itens removidos e inseridos serão atualizados.

Se não houver espaço para inserir uma mensagem, então é retornado erro (*errQUEUE_FULL*).

#### 5.4.2.4 *FromISR*

Existem outras funções que não devem ser chamadas dentro de uma interrupção, mas sim suas variantes. Essas funções sempre serão acompanhadas por *FromISR*. Por exemplo, ao invés de chamar *xQueueGenericReceive*, passando como parâmetro a variável *xJustPeeking* (para ver o dado da fila, mas não retirá-lo dela), chama-se *xQueuePeekFromISR*.

# 6 CONCLUSÃO

Analisando a estrutura dos recursos, pode-se perceber que o código em si é bastante compacto (por exemplo semáforos que são na verdade filas). Essa economia de código (e as vezes até remoção, como a proteção de memória que para este *port* não existe) é extremamente necessária em arquiteturas minimalistas como o Arduino Uno. Isso acarreta em mais espaço para o que o escritor do código necessita em seu projeto. Entretanto, pode-se perceber também que o uso do FreeRTOS no Arduino Uno é bastante específico, pois não existe a possibilidade de fazer muitas tarefas diferentes com ele (devido a sua pequena memória) e nem tarefas bastante complexas (tendo em vista o modo como os *timers* e filas trabalham, somado com a pequena memória e a frequência de seu *clock*).

Com o presente trabalho, é possível analisar melhor o projeto e tomar a decisão de uma forma mais clara se o FreeRTOS é uma opção viável e se ele conseguirá suprir a demanda. Suas funções melhores detalhadas também permitem ao escritor alterar alguma parte do código mais facilmente (caso necessite) e entender o seu comportamento, para verificar qual recurso é o melhor para determinada aplicação. Suas estruturas detalhadas servem para definir a quantidade de memória usada ao final do projeto e comparar se há a necessidade de usar outro modelo minimalista ou se o FreeRTOS tem capacidade suficiente de suprir essa demanda.

Ademais, este trabalho mostra que existem situações em que o FreeRTOS é suficiente e efetivo para algumas necessidades (tarefas simples e que não necessitam de muita memória) e portanto é uma biblioteca interessante para ser usada em conjunto ao Arduino Uno.

# 7 ANEXOS

## 7.1 ESTRUTURA DE FILA (QUEUE_T)

Econtrada em *queue.c*. Possui no total 31 *bytes* (mais 2 pelo *malloc*, totalizando 33). Foram removidas variáveis que são declaradas apenas em casos específicos (tais como se usar conjuntos de filas, que não são por padrão).

```
typedef struct QueueDefinition
{
  (1B) int8_t *pcHead
  (2B) int8_t *pcTail
  (2B) int8_t *pcWriteTo
  (1B) union{
    int8_t *pcReadFrom
    UBaseType_t uxRecursiveCallCount
  }
  (9B) List_t xTasksWaitingToSend
  (9B) List_t xTasksWaitingToReceive
  (1B) UBaseType_t uxMessagesWaiting
  (1B) UBaseType_t uxLength
  (1B) UBaseType_t uxItemSize
  (1B) UBaseType_t xRxLock
  (1B) UBaseType_t xTxLock
} xQUEUE;
```

## 7.2 ESTRUTURA DE TIMER (*TIMER_T)*

Encontrada *timers.c*. Possui no total 19 *bytes* (mais 2 pelo *malloc*, totalizando 21). Foram removidas variáveis que são declaradas apenas em casos específicos (tais como se usar conjuntos de filas, que não são por padrão).

```
typedef struct tmrTimerControl
{
  (2B) const char *pcTimerName;
  (10B) ListItem_t xTimerListitem;
  (2B) TickType_t xTimerPeriodInTicks;
  (1B) UBaseType_t uxAutoReload;
  (2B) void *pvTimerID;
```

```
  (2B) TimerCallbackFunction_t pxCallbackFunction;
} xTIMER;
```

## 7.3 ESTRUTURAS DE LISTA

### 7.3.1 *List_t*

Econtrada em *list.h*. Possui no total 9 *bytes*. Foram removidas variáveis que são declaradas apenas em casos específicos (tais como se usar conjuntos de filas, que não são por padrão).

```
typedef struct xLIST
{
  (1B) uxNumberOfItems;
  (2B) ListItem_t * configLIST_VOLATILE pxIndex;
  (6B) MiniListItem_t xListEnd;
} List_t;
```

### 7.3.2 *ListItem_t*

Econtrada em *list.h*. Possui no total 10 *bytes*. Foram removidas variáveis que são declaradas apenas em casos específicos (tais como se usar conjuntos de filas, que não são por padrão).

```
typedef struct xLIST_ITEM
{
  (2B) configLIST_VOLATILE TickType_t xItemValue;
  (2B) struct xLIST_ITEM* configLIST_VOLATILE pxNext;
  (2B) struct xLIST_ITEM* configLIST_VOLATILE pxPrevious;
  (2B) void *pvOwner;
  (2B) void *configLIST_VOLATILE pvContainer;
} ListItem_t;
```

### 7.3.3 *MiniListItem_t*

Econtrada em *list.h*. Possui no total 6 *bytes*. Foram removidas variáveis que são declaradas apenas em casos específicos (tais como se usar conjuntos de filas, que não são por padrão).

```
typedef struct
{
  (2B) configLIST_VOLATILE TickType_t xItemValue;
  (2B) struct xLIST_ITEM* configLIST_VOLATILE pxNext;
  (2B) struct xLIST_ITEM* configLIST_VOLATILE pxPrevious;
} MiniListItem_t
```

## 7.4 TESTE DE TAMANHOS

### 7.4.1 Código de teste

Para realizar o teste de tamanhos, foi usado o seguinte código:

```
#include <Arduino_FreeRTOS.h>
#include <timers.h>
#include <queue.h>


void vPrintTimer(){
  for(;;) Serial.print("Tarefa Desnecessaria");
}

void setup() {
  int tam;
  Serial.begin(9600);
  // wait for Leonardo
  while(!Serial) {}

  Serial.print("Tamanho do(a) ListItem_t: ");
  Serial.println(sizeof(ListItem_t));
  Serial.print("Tamanho do(a) List_t: ");
  Serial.println(sizeof(List_t));
  Serial.print("Tamanho do(a) MiniListItem_t: ");
  Serial.println(sizeof(MiniListItem_t));
  Serial.print("Tamanho do(a) TimerCallbackFunction_t: ");
  Serial.println(sizeof(TimerCallbackFunction_t));

  // start FreeRTOS
  tam = xQueueCreate(1, 50);
```

```
    Serial.print("Tamanho do(a) Queue_t: ");
    Serial.println(tam);

    tam = xTimerCreate("Timer", NULL, pdFALSE, NULL, vPrintTimer );
    Serial.print("Tamanho do(a) Timer_t: ");
    Serial.println(tam);
    // should never return
    Serial.println(F("Die"));
    while(1);
  }

  void loop() {
    // put your main code here, to run repeatedly:
  }
```

### 7.4.2 Alterações em código fonte do FreeRTOS

Para o código acima funcionar, foi necessário alterar os seguintes trechos de códigos da biblioteca do FreeRTOS:

```
//Alterado o retorno para int e agora a função retorna o tamanho de Queue_t;
int xQueueGenericCreate( const UBaseType_t uxQueueLength,
  const UBaseType_t uxItemSize, const uint8_t ucQueueType )
{
  Queue_t *pxNewQueue;
  size_t xQueueSizeInBytes;
  QueueHandle_t xReturn = NULL;
  return sizeof(Queue_t);
}

//Alterado o retorno da função para int;
int xQueueGenericCreate( const UBaseType_t uxQueueLength,
const UBaseType_t uxItemSize, const uint8_t ucQueueType ) PRIVILEGED_FUNCTION;

//alterado o retorno para int e agora a função retorna o tamanho de Timer_t;
int xTimerCreate( const char * const pcTimerName,
  const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload,
  void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction )
{
  Timer_t *pxNewTimer;
  return sizeof(Timer_t);
```

```
}

//Alterado o retorno da função para int;
int xTimerCreate( const char * const pcTimerName,
  const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload, void * co
```

## 7.5 OUTRAS INFORMAÇÕES

Outras informações sobre funções e estruturas pesquisadas podem ser encontradas no trabalho de Chabatura F (2018).

# REFERÊNCIAS


1 AG, Arduino. **What is Arduino?** [S.l.: s.n.], 2018. https://www.arduino.cc/. Accesso em: 15-06-2018.

2 BARRY, Richard. Mastering the FreeRTOS Real Time Kernel-a Hands On Tutorial Guide. **Real Time Engineers Ltd**, 2016.

3 _____. **Source Organization**. [S.l.: s.n.], 2018. https://www.freertos.org. Accesso em: 15-06-2018.

4 BUONOCUNTO, Pasquale et al. ARTE: arduino real-time extension for programming multitasking applications. In: ACM. PROCEEDINGS of the 31st Annual ACM Symposium on Applied Computing. [S.l.: s.n.], 2016. p. 1724–1731.

5 CHABATURA F, Fassini L. T. **Descrições de Funções e Estruturas da Versão 8.2.3 do FreeRTOS para o Arduino Uno.** [S.l.: s.n.], 2018. Zenodo. http://doi.org/10.5281/zenodo.1580268. Accesso em: 5-11-2018.

6 ISO. **ISO/IEC 9899:TC3**. [S.l.: s.n.], 1999. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf. Accesso em: 5-11-2018.

7 PEREIRA, Jorge et al. Co-designed freertos deployed on fpga. In: IEEE. COMPUTING Systems Engineering (SBESC), 2014 Brazilian Symposium on. [S.l.: s.n.], 2014. p. 121–125.

8 SACHA, Krzysztof M. Measuring the real-time operating system performance. In: IEEE. REAL-TIME Systems, 1995. Proceedings., Seventh Euromicro Workshop on. [S.l.: s.n.], 1995. p. 34–40.

9 STANKOVIC, John A; RAJKUMAR, Raj. Real-time operating systems. **Real-Time Systems**, Springer, v. 28, n. 2-3, p. 237–253, 2004.

10 STEVENS, Philip. **A FreeRTOS Library for all Arduino AVR Devices (Uno, Leonardo, Mega, etc)**. [S.l.: s.n.], 2018. https://github.com/feilipu/Arduino_FreeRTOS_Library. Accesso em: 15-06-2018.

11 TANENBAUM, Andrew S; MACHADO FILHO, Nery. **Sistemas operacionais modernos**. [S.l.]: Prentice-Hall, 1995. v. 3.